CS202 Data Structures

Assignment 2

On building your own search engine

Due: 11pm on Friday, 10th March, 2017 (on LMS)

Late Submission Policy

This assignment can be submitted till 11pm on Monday, 13th March, 2017 with a 10% penalty per day.

In this assignment, you will implement a small-scale *search engine* using different data structures. In particular, you will implement this search engine using *trees*, *binary search trees,* and *AVL trees*. This will help you in appreciating the tradeoffs (e.g., running times vs. memory requirements) associated with using different data structures.

This assignment is more intense than the first assignment and contains four tasks so start early!

The course policy about plagiarism is as follows:

1. Students must not share actual program code with other students.
2. Students cannot copy code from the Internet.
3. Students must be prepared to explain any program code they submit.
4. Students must indicate any assistance they received.
5. All submissions are subject to automated plagiarism detection.

Students are strongly advised that any act of plagiarism will be reported to the Disciplinary Committee.

Task-1: [15 Points]

In Task-1, you will write a program that searches for files on your personal computer efficiently (*and obviously without using the Windows search feature* ☺). Your program will read the contents of a portion of your Windows directory from a text file. Your program will save the directory contents in a tree data structure. Then, the user of your program can search for exact matches of any file or directory name.

The basic layout of this directory tree class is provided to you in *tree.h.* You may add other functions to this header file.

Input Format:

The input text file may contain a directory list in the following format:

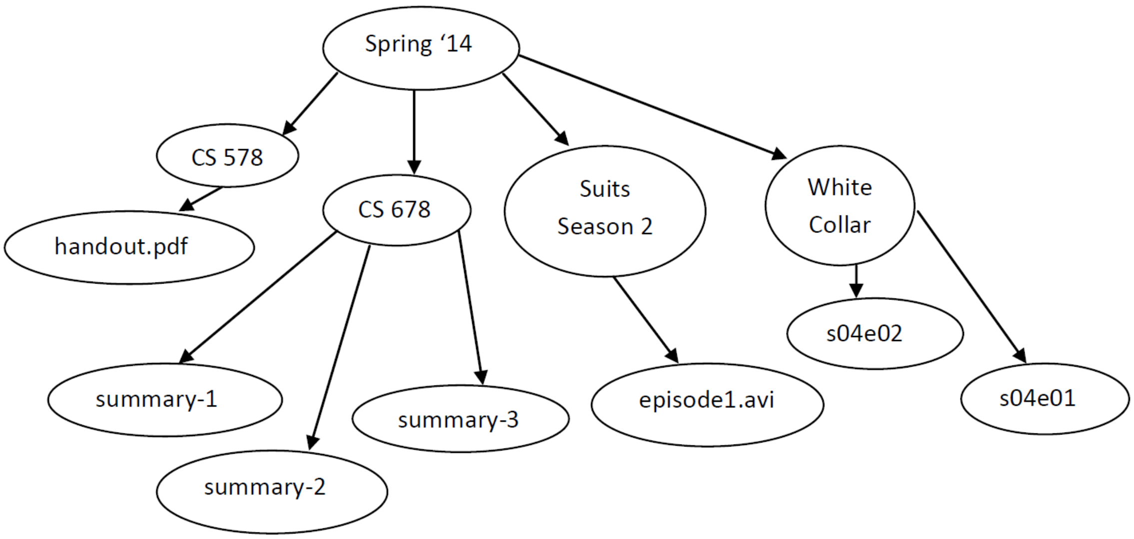
```
\Spring '17\CS 578
\Spring '17\CS 678
\Spring '17\Suits Season 2
\Spring '17\White Collar
\Spring '17\CS 578\handout.pdf
\Spring '17\CS 678\summary-1.docx
\Spring '17\CS 678\summary-2.docx
\Spring '17\CS 678\summary-4.docx
\Spring '17\CS 678\summary-5.docx
\Spring '17\Suits Season 2\episode1.avi.
\Spring '17\Suits Season 2\torrent.txt.txt
\Spring '17\White Collar\s04e01.avi
\Spring '17\White Collar\s04e02.avi
```

For simplicity, in this assignment we assume that there are no duplicate file names or directory names.

As part of the assignment, you will have to first generate a text file of all the directories on your own PC. This can be easily done by opening the command prompt on your windows machine and then typing the command "**dir /s /b > input.txt**"**.** This will create a file named input.txt in the current directory. Transfer this file to your mars account or Linux machine so that your program may use it.

Directory Tree:

Below is how the tree data structure corresponding to the above sample input looks like on paper. You should program the tree by using the sibling pointers as taught in the class.



The constructor for the tree class expects the filename as the input parameter. The program expects the specified file to contain the directory contents as mentioned earlier. Inside the constructor, it will read the directories from that file and load into memory in a tree data structure. Your tree should support two operations:

1) **Locate**: Takes the name of the file to search for, and returns the complete path of the file in a vector i.e., If the user enters “**s04e02**” the returned vector would contain **“s04e02” “White Collar”** “**Spring'14”** in this order (root folder is the last element of the vector). You should also print out the time taken for the search.

2) **Lowest Common Ancestor:** Given names of two different files or directories, your function should return the *Lowest Common Ancestor* (LCA) for them. For example, the LCA for “**summary-2**”and “**summary-3**” is “**CS 678**”.

Task-2: [35 points]

In this task, you will write a program that reads key-value pairs from an input file and constructs a corresponding **Binary Search Tree (BST)**. The nodes in the BST are to be arranged according to the relative magnitude of the keys. The basic layout of this BST class is provided to you in *bst.h*

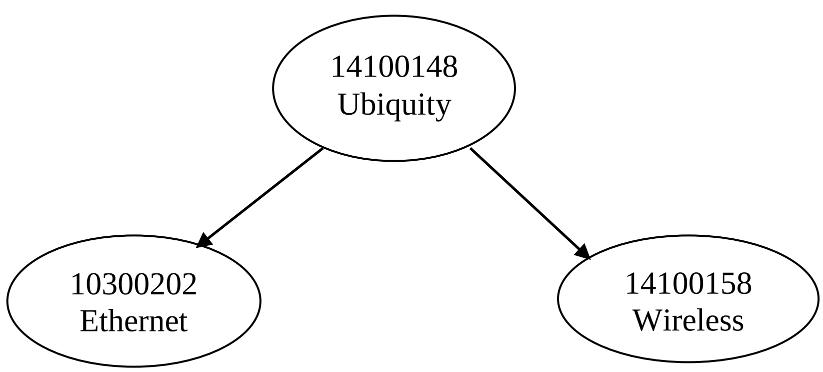
You are provided with two (test) input files of different sizes that contain key-value pairs: **names.txt** and **pairs.txt**. While names.txt has ~180 entries, the input file **pairs.txt** has 1 million entries. It may be helpful to start testing your program on **names.txt** as it is a short test file.

Input Format:

Each line includes a value (a string) followed by its unique key (which can be a string, integer, or a float).'~' Marks the end of a value. For example:

```
Ubiquity~ 14100148
Wireless~ 14100158
Ethernet~ 10300202
```

The corresponding BST would look like:



Requirements:

1. Your BST should support **insertion** of a **key-value pair**. Insertion should not violate any properties of a BST and should be with respect to the key. All the keys in pairs.txt are odd numbers. So, in order to test your insert function, you can run your program and have it insert any even number.
2. BST should support **deletion** of a **key-value pair**. Deletion should properly free the memory and rearrange the tree to maintain its properties.
3. BST should support **search** operation, given a **key** and should return the corresponding value.

4. BST should support **replacement** of a particular key by a new key. This operation should update the position of the node in the BST. You should not allow the user to update a key that is already in use.
5. BST should have a function that returns the **height** of the tree.
6. BST should have a function that prints the execution time taken by each function.

A Sample Run of your program would look like:

Please enter a filename:

> Pairs.txt

File loaded successfully. 4745232 key/value pairs loaded from the file in 0 seconds and 548 microseconds.

- **Press 1 to Insert another key/value Pair**
- **Press 2 to search for a value using its key**
- **Press 3 to delete a key/value pair**
- **Press 4 to update the key of an existing value**
- **Press 5 to see the current height of the Tree**

2

Please enter a key to search:

> 35

Result found in 0 seconds and 4431 microseconds.

Value corresponding to your key is:

MNMBLQVJJFRPWVVAIAZM

Task-3: [30 points]

Repeat Task-2 but this time store all data extracted from **pairs.txt (or names.txt)** in an **AVL tree**.

After prompting the user for the text file's name, the program should also prompt the user to choose the data structure to use as follows:

- **Please Choose Data Structure:**
- **1) BST**
- **2) AVL**

Requirements are the same as those for BST except that the Delete and Update operations are for **extra credit**.

The basic layout of this AVL class is provided to you in *avl.h*

Hint: You will need to implement all the rotation operations. Further, you will need to maintain the height attribute for each node in the AVL tree to make your rotations efficient (*finding the height for a node from scratch will take O(number of nodes) each time, which will make our rotation operations practically useless*).

Printing ExecutionTimes:

To print out the execution time of a function, you should call *startTimer()* before you make call to your function and call *stopTimer*() when you return from your function call. This will print out the execution time of your function. The *startTimer()* and *stopTimer*() are defined inside the *time.h* file that has been provided to you.

Task-4: [20 points]

In this task, you will write a program to store the roll number and aggregate score of CS202 students in an AVL tree. The nodes in the AVL should be arranged according to the relative magnitude of aggregate scores. The basic layout of this program is given to you in the file avl2.h.

You are provided with an input file ("input_avl.txt") containing the roll numbers and aggregate scores. You are required to read the input file and store the information in an AVL tree.

Input Format:

The input file will be in the following format:

19100321 58.7

19100019 96.5

19020301 78.9

...

Each line contains the information of one student where the roll number and aggregate score will be separated by a white space.

Assumptions:

Each student has a unique aggregate score. When inserting the information of a student, you need to make sure there is no other student with that aggregate score.

Requirements:

1. Your AVL tree implementation should support the search operation (i.e., given a score it should return the corresponding node or NULL)

2. Your implementation should have a function getTopFive to find the highest five aggregates in the class. This function should return an array where the first element has the highest aggregate, second element has the second highest aggregate and so on

3. Your AVL implementation should have a function getAverage that returns the average score of all students

4. You AVL implementation should have a function displayAll to display the roll number and aggregate scores in decreasing order of aggregate scores